(NASA-CR-174151) A FAULT-TOLERANT SCHEDULING PROBLEM (Illinois Univ.) 26 p

N84-72600

Unclas
00/61 12118

80-1709

UIUCDCS-R-80-1010

UILU-ENG 80 1709

A FAULT-TOLERANT SCHEDULING PROBLEM

by

Arthur L. Liestman
Roy H. Campbell

February 1980

UIUCDCS-R-80-1010

A FAULT-TOLERANT SCHEDULING PROBLEM

by

Arthur L. Liestman
Roy H. Campbell'

February 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

This project is funded in part by NASA Grant NSG 1471.

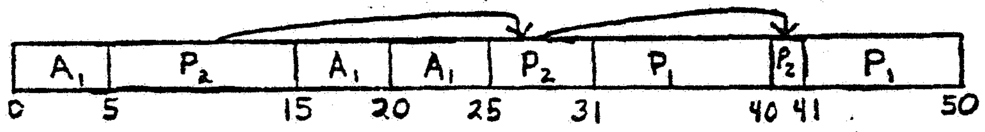
1 Introduction.

A real-time system provides a service which meets a set of specifications including real-time constraints. It is desirable to guarantee reliability in a real-time system. Reliability is a measure of how well the system conforms to its specifications. One technique used to improve reliability is fault-tolerance which incorporates redundancy into the system design. This redundancy is combined with error detection and error confinement techniques to prevent isolated failures from causing system failure. A deadline mechanism [Campbell, et al., 79] has been proposed to provide fault-tolerance in real-time systems. In this mechanism two independent algorithms are provided for each service subject to a deadline. An algorithm is presented here which produces a fault-tolerant schedule for such a real-time system.

Consider a scheduling problem in which a time-shared single-processor computing system is to execute a set of jobs each of which consists of a sequence of periodic requests. That is, each job periodically demands a response within a certain time interval. A further property of the proposed system is that each job's request period is a multiple of the next smallest request period. Such a system is termed simply periodic. Let $J=\{J_1, J_2, \ldots, J_r\}$ denote a set of jobs with periodic requests. T_i denotes the request period, P_i denotes the computation time of the primary, and A_i denotes the computation time of the alternate for job J_i $i=1, 2, \ldots, r$. Assume that $A_i < P_i$ for $i=1, 2, \ldots, r$. The level of the job J_i is i. The jobs are ordered such that $m_i T_i = T_{i+1}$ for some positive integer $m_i > 2$ for $i=1, 2, \ldots, r-1$.

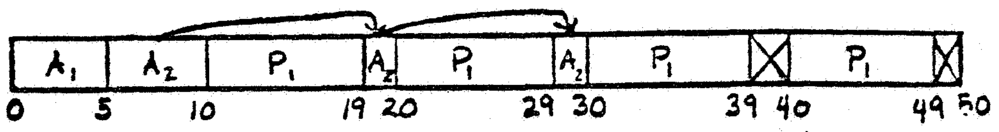
The deadline mechanism provides two algorithms for each service. The primary algorithm produces a good quality service but is subject to timing errors which are precisely defined in [Campbell, et al., 79]. The alternate algorithm produces an acceptable response and by definition is not subject to timing errors. The response to a request can consist of the completed execution of either the primary or the alternate algorithm. The best schedule is obviously one that successfully executes the primary algorithm for each request, but due to possible primary failures this is not always possible.

Given this information about the set of jobs a schedule for the execution of the responses can be specified. The deadline of a request is the time at which the next request of the same job arrives. Scheduling a set of jobs with simply periodic requests denotes specification of which alternate or primary is to be executed at every time instant. A schedule is feasible if all requests will be satisfied before their deadlines. The execution of an alternate or primary can be interrupted. Consider the following example: J_1, J_2 denote jobs with A_1=5, P_1=9, T_1=10, A_2=7, P_2=17, and T_2= 50. A schedule is described by a timing diagram such as the following for the above example:



The execution of P_2 is divided into three sections which are scheduled in the intervals 5-15, 25-31, and 40-41.

Due to the nature of the primary and alternate algorithms it is desirable to maximize the number of primaries executed while still ensuring that all deadlines are met. In the above example, two P_1's and one P_2 are executed during the period of J_2. As the following schedule illustrates, the number of primaries executed in this example can be improved:



In this schedule four P_1's are executed and idle time is scheduled during the intervals 39-40 and 49-50. This is the largest number of primaries which could be executed in one period of J_2.

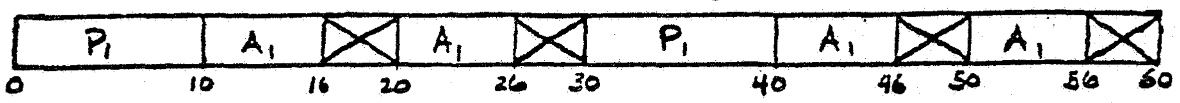
Several algorithms to create schedules for simplified versions of the real-time system are developed below. The scheduling algorithm of Chapter 2 creates a static schedule for the period T_r which maximizes the number of primaries scheduled. The two algorithms described in Chapter 3 are modifications of the Chapter 2 algorithm. The first algorithm in Chapter 3 creates a static fault-tolerant schedule for the period T_r. In this schedule the deadline is met by the alternate if a scheduled primary fails. The number of attempted primaries in this schedule is maximized among those schedules which guarantee that all the deadlines will be met. The second algorithm in Chapter 3 creates a new schedule whenever idle time is made available during the execution of the fault-tolerant schedule due to successful primaries. The combination of these two algorithms yields a dynamic scheduling algorithm maximizing the number of primaries scheduled while guaranteeing fault-tolerance. Chapter 4 describes a tree of schedules which may be precomputed and used to implement the actions of this dynamic algorithm in the real-time system.

2 A Scheduling Algorithm.

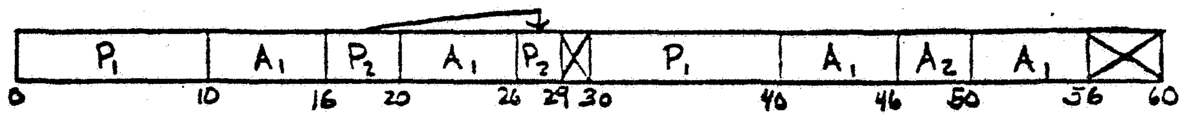
Given a set of jobs $J=\{J_1, J_2, \ldots, J_r\}$ a schedule can be created for the period T_r which will maximize the number of primaries executed given A_i, P_i, and T_i for $i=1, 2, \ldots, r$. For the moment the possibility of primary failure is ignored. An optimal schedule is feasible and has the maximum number of primaries scheduled among all feasible schedules. The schedule is produced in three phases. First a set of counters NP(i) is produced which indicates the number of primaries of each level i included in an optimal schedule. The second phase is the production from each NP(i) of a list of primaries and alternates to be executed for each level. The third phase is the creation of the schedule from these lists.

The values NP(i) for $J = \{J_1, J_2, \ldots, J_r\}$ are created by iteratively creating the values of NP(i) for the sets $\{J_1\}$, $\{J_1, J_2\}$, ..., $\{J_1, J_2, \ldots, J_r\}$. In each case the counters represent a schedule for the period of the highest level job. For the jobs $\{J_1, J_2, \ldots, J_i\}$ the schedule is created for the period T_i by concatenating m_{i-1} copies of the schedule for $\{J_1, J_2, \ldots, J_{i-1}\}$ and then modifying the resulting schedule.

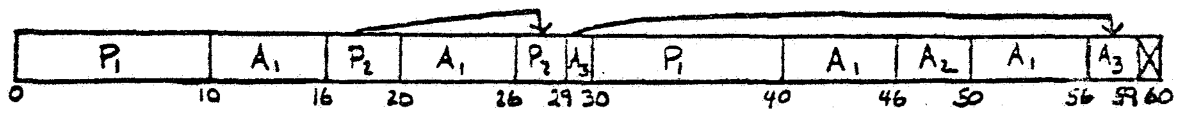
The entire schedule is constructed from the NP(i) values. Consider the following example. Let $A_1=6$, $P_1=10$, $T_1=10$, $A_2=4$, $P_2=7$, $T_2=30$, $A_3=4$, $P_3=10$, $T_3=60$, NP(1)=2, NP(2)=1, and NP(3)=0. The algorithm distributes the lower level primaries within the larger periods. Since there are 2 P_1s in the entire schedule and since there are 2 T_2s in T_3 each T_2 includes a P_1. Within each T_2 there are 3 T_1s so the P_1s must be augmented with A_1s. The algorithm produces tuples which represent lists of primaries and alternates for each level. In the above example the level 1 tuple, (100100), corresponds to scheduling the sequence $P_1A_1A_1P_1A_1A_1$ at the level 1 request times:



The level 2 tuple, (10), indicates that P_2 and A_2 are scheduled in the remaining idle time of the T_2 period:



The level 3 tuple is (0) indicating that A_3 fills the remaining idle time:



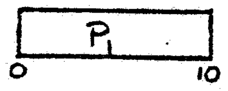
Algorithm 2.1 constructs an optimal schedule. The algorithm uses a list called the diff list. This list keeps the level numbers i for those values of i with NP(i)>0 sorted in decreasing order of $P_i - A_i$. Initially the list is empty. The notation $\lceil x \rceil$ denotes the smallest integer not less than x. The algorithm follows:

Algorithm 2.1

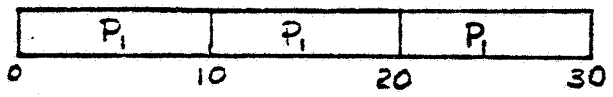
```
program SCHEDULE (* CREATE AN OPTIMAL SCHEDULE FOR J *)

  procedure MODIFY(ALT,PRIM,LEVEL)
  (* ADD A LEVEL I RESPONSE TO AN OPTIMAL LEVEL I-1 SCHEDULE
       OVER THE LEVEL I PERIOD *)

  (* STEP 1- CREATE SPACE FOR ALT *)
  while idle time < ALT
    begin
      let d be the first level number in the diff list
      k := $\lfloor (ALT - idle\ time)/(P_d - A_d) \rfloor$
      if k > NP(d)
      then begin
         idle time := idle time + NP(d) * $(P_d - A_d)$
         NP(d) := 0
         remove d from the diff list
      end
      else begin
         idle time := idle time + k * $(P_d - A_d)$
         NP(d) := NP(d) - k
      end
  end (* OF STEP 1 *)


  if PRIM < idle time
  then begin

     (* STEP 2a- SCHEDULE PRIMARY IF IT FITS *)
     idle time := idle time - PRIM
     NP(LEVEL) := 1
     insert LEVEL into the diff list
  end (* OF STEP 2a *)

  else begin
     if diff list is not empty
     then begin

         (* STEP 2b - EXCHANGE PRIMARIES IF IDLE IS GAINED *)
        let d be the first level number in the diff list
        if PRIM - ALT < $P_d - A_d$
        then begin
           idle time := idle time + $P_d - A_d$ - PRIM
           NP(d) := NP(d) - 1
           if NP(d) = 0 then remove d from diff list
           NP(LEVEL) := 1
           insert LEVEL into diff list
        end (* OF STEP 2b *)
      else begin

          (* STEP 2c- SCHEDULE ALTERNATE *)
          idle time := idle time - ALT
      end (* OF STEP 2c *)
  end
end (* of MODIFY *)


begin  (* MAIN PROGRAM *)

  (* PHASE 1 --- LOOP TO CREATE NP(i) VALUES *)
  idle time := $T_1$
  $m_0$ := 1

  for i := 1 to r do
    NP(i) := 0
    for j := 1 to i-1 do
      NP(j) := NP(j) * $m_{i-1}$
      end
    idle time := idle time * $m_{i-1}$
    MODIFY($A_i$,$P_i$,i)
    end

(* PHASE 2 --- CONVERT COUNTERS INTO SCHEDULE *)
consider each NP(i) a 1-tuple
for i := 1 to r-1 do
  for j := r-1 downto i do
     for each element h of the $i^{th}$ tuple do
        k := h mod $m_j$

        n := k div $m_j$

        replace h with an $m_j$-tuple.  the first k elements of the

            tuple are n+1 and the remaining elements are n.
        end
     end
  end
```

```
 (* PHASE 3 --- SPECIFY THE SCHEDULE *)
for i := 1 to r do
  for each element of the level i tuple do

      if j^th element of level i tuple = 1
      then  schedule primary for level i in the first
               $P_i$ units of idle time after $(j-1)T_i$

      else  schedule alternate for level i in the first
              $A_i$ units of idle time after $(j-1)T_i$
      end
   end
end (* OF MAIN PROGRAM *)
```

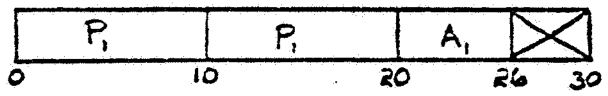
Let J_1, J_2, and J_3 be jobs such that $A_1=6$, $P_1=10$, $T_1=10$, $A_2=4$, $P_2=7$, $T_2=30$, $A_3=4$, $P_3=10$, and $T_3=60$. Each time an NP(i) value changes the corresponding schedule is given below. On the first call to MODIFY P_1 is scheduled since $P_1=10 \leq$ idle time=10:



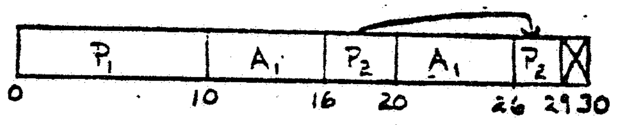
When i=2, 3 copies of the above schedule are concatenated to give:



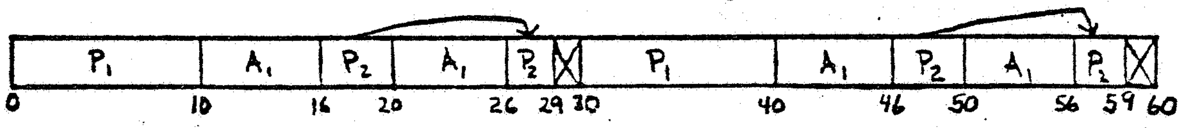
Step 1 of MODIFY changes a P_1 to A_1:



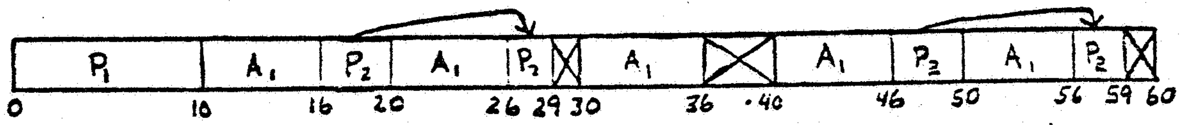
Since $P_2=7 >$ idle time=4 and $P_2-A_2=3 < P_d-A_d=4$, one more P_1 is changed to A_1 and P_2 is scheduled:



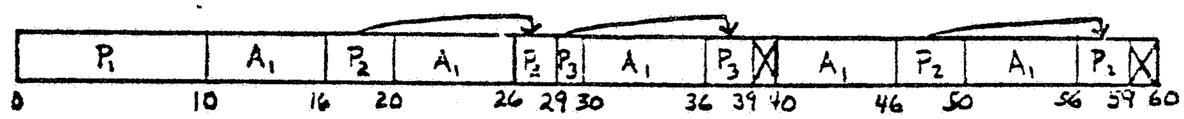
When i=3, 2 copies of the above schedule are concatenated to give:



Step 1 of MODIFY changes a P_1 to A_1:



Since P_3=10 > idle time=6 and P_3-A_3=6 > P_d-A_d=4, A_3 is scheduled:



Theorem 2.1: The schedule produced by Algorithm 2.1 is optimal and has as much idle time scheduled as any optimal schedule.

Proof: (by induction on r)

For r=1, the algorithm schedules P_1 if $P_1 < T_1$ and schedules A_1 otherwise. This is clearly optimal and the idle time is maximized among optimal schedules since all optimal schedules have the same amount of idle time.

Assume that the algorithm produces an optimal schedule with maximum idle time for any set of p jobs.

Consider the set of jobs J = $\{J_1, J_2, \ldots, J_{p+1}\}$. The first p iterations of the algorithm produce an optimal schedule for J′ = $\{J_1, J_2, \ldots, J_p\}$ with maximal idle time. Concatenate m_p copies of this schedule and call the resulting schedule S. Let t be the number of primaries in S. Clearly S is an optimal schedule for the jobs in J′ over the period T_{p+1}. Either A_{p+1} or P_{p+1}

is to be added to the schedule.

It is desirable to maximize the number of primaries in the final schedule. The number of primaries contributed by the jobs in J' cannot exceed t. At least A_{p+1} units of idle time are needed to schedule a response for J_{p+1}. If the idle time in S is less than A_{p+1} then there is no feasible schedule for J with t primaries for the jobs in J'. Thus, some of the primaries must be changed to alternates so that either A_{p+1} or P_{p+1} can be scheduled. By changing those primaries with the largest diff (P_i-A_i) values first, the number of primaries changed is minimum and among such changes the idle time, when A_{p+1} is scheduled, is maximized. Thus if A_{p+1} is scheduled an optimal solution for J has been found.

There are two cases under which P_{p+1} might be scheduled instead of A_{p+1}. First, if P_{p+1} fits in the time allotted for A_{p+1} plus the remaining idle time then clearly this solution is optimal since it includes one more primary than the solution with A_{p+1}. Second, if a single P_j for $j<p+1$ could be converted to A_j so that P_{p+1} fits into the time allotted for A_{p+1} plus the remaining idle time plus $P_j - A_j$ and the resulting idle time is greater than the idle time in the solution with A_{p+1}. In this case the idle time is increased and the number of primaries remains the same. Among such solutions, an optimal solution is one such that P_j-A_j is maximum thus leaving the largest idle time in the solution for J.

Let $M_i = m_1m_2\ldots m_i$. Let $M_0 = 1$.

Theorem 2.2: Algorithm 2.1 creates a schedule for $O(M_{r-1})$ jobs in $O(M_{r-1})$ time.

Proof: Consider the number of jobs scheduled. Clearly there are $m_i m_{i+1} \cdots m_{r-1}$ requests for J_i for $i<r$ and 1 request for J_r. The total number of requests is $\sum_{i=0}^{r-1} (M_{r-1}/M_i)$. Since $m_i \geq 2$ for all i then $M_i \geq 2^i$, thus $M_{r-1} < \sum_{i=0}^{r-1} (M_{r-1}/M_i) < 2M_{r-1}$. Thus $O(M_{r-1})$ jobs are scheduled.

Consider the time required to create the NP(i) values. The main program takes $O(r^2)$ steps to initialize the counters and copy the intermediate solutions. There are r calls to MODIFY. On the ith such call at most i iterations of Step 1's while loop are possible. Each of these iterations takes constant time. On each call to MODIFY exactly one of the Steps 2a, 2b, and 2c is executed. Step 2c requires constant time. Each of the other steps may involve an insertion into a list of fewer than i elements but otherwise they each require constant time. The insertion requires O(i) steps. Thus the calls to MODIFY require $O(r^2)$ steps. Therefore $O(r^2)$ steps are needed to create the NP(i) values.

Transforming NP(i) into the tuple requires $m_{i+1} m_{i+2} \cdots m_{r-1} = M_{r-1}/M_i$ steps. Summing over the values of i as before we get $O(M_{r-1})$. Similarly, converting the tuples into the schedule requires $O(M_{r-1})$ time. Thus Algorithm 2.1 requires $O(M_{r-1})$ time.

Given a set of jobs J, an algorithm for construction of an optimal schedule for J has been given. The algorithm builds the schedule iteratively, one level at a time. Only a few counters are required until the final schedule is to be written out. At this point each of the counters yields a sequence of primaries and alternates to be executed. The schedule can then be constructed from this sequence.

3 A Fault-tolerant Scheduling Algorithm.

Algorithm 2.1 produces a schedule to maximize the number of primaries executed. The primary algorithm is susceptible to timing errors. In some cases the actual execution time of the primary is not known in advance. The value P_i may be the expected execution time or the minimum execution time of the primary. The use of Algorithm 2.1 with such primaries can clearly lead to failure to meet the real-time constraints. In order to insure a fault-tolerant schedule every request for J_i must be fulfilled by executing either the alternate or the primary for level i.

It is desirable to guarantee that the failure of P_i does not inhibit execution of A_i before the deadline. The following changes to Algorithm 2.1 produce Algorithm 3.1:

1. The call MODIFY(A_i,P_i,i) is replaced by MODIFY(A_i,P_i+A_i,i).
2. Every occurrence of P_d-A_d in MODIFY is replaced by P_d.
3. In Phase 3, 'primary' is replaced by 'primary followed by alternate' and P_i, P_r are replaced by P_i+A_i and P_r+A_r, respectively.

This algorithm creates a schedule maximizing the number of primaries scheduled with the additional constraint that whenever a primary is scheduled its alternate is scheduled to follow it.

A schedule is f-t feasible if all requests will be satisfied before their deadlines even if no primary algorithms succeed. A schedule is f-t optimal if it is f-t feasible and has the maximum number of primaries scheduled among all f-t feasible schedules.

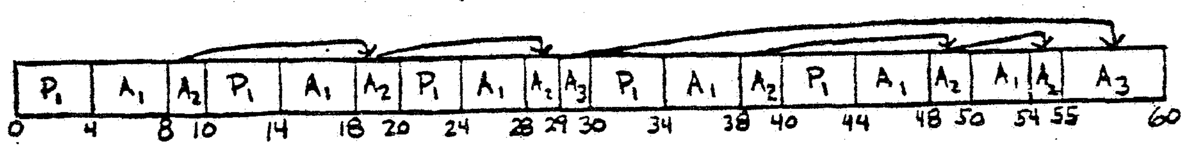
Theorem 3.1: The schedule produced by Algorithm 3.1 is f-t optimal and has as much idle time as any f-t optimal schedule.

Proof: follows easily from Theorem 2.1.

Theorem 3.2: Algorithm 3.1 creates a f-t schedule for $O(M_{r-1})$ jobs in $O(M_{r-1})$ time.

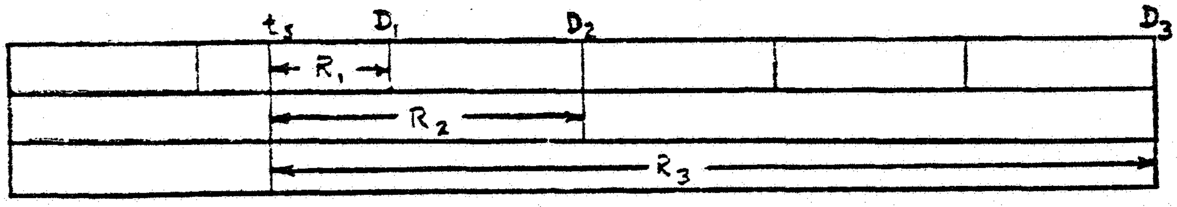
Proof: follows from Theorem 2.1.

Let J_1, J_2, and J_3 be jobs such that $A_1=4$, $P_1=4$, $T_1=10$, $A_2=5$, $P_2=7$, $T_2=30$, $A_3=6$, $P_3=8$, and $T_3=60$. The f-t scheduling algorithm produces the following schedule:



As the scheduled jobs are executed assume that at time 4, P_1 fails to complete. A_1 is then executed and the deadline for J_1 at time 10 is met when A_1 completes at time 8. The 2 units from 8 to 10 are used to begin execution of A_2. At time 10, P_1 begins to execute and succeeds at time 14. The request by J_1 has been satisfied and thus the time allocated to A_1 in the interval 14-18 can now be set to idle. Algorithm 3.2 can be used to reallocate this wasted time.

Assume that P_s succeeds at time t_s. A new schedule for the interval t_s to T_r is to be created with the maximum number of primaries scheduled. Some parts of alternates and primaries on other levels may have already been executed. Consider the following representation of the periodic structure:



Define EXA_i to be the number of time units of A_i already executed during the current J_i period when P_s succeeds at time t_s. Similarly, EXP_i is defined to be the number of time units of P_i already executed. Note that EXA_i must be updated by the system at run time. The next J_i deadline after t_s, called D_i, can be computed by: $D_i = \lceil t_s/T_i \rceil * T_i$. Let $R_i = D_i - t_s$ denote the remaining time before the next J_i deadline. When P_s succeeds, compute D_i and R_i for each level $i \neq s$. Between R_i and D_i a response to the request for J_i must be scheduled if the request has not already been satisfied. This response may be either a primary followed by an alternate or just an alternate. The times required for these responses are $P_i+A_i-EXA_i-EXP_i$ and A_i-EXA_i respectively. From D_i to D_r responses are scheduled as before.

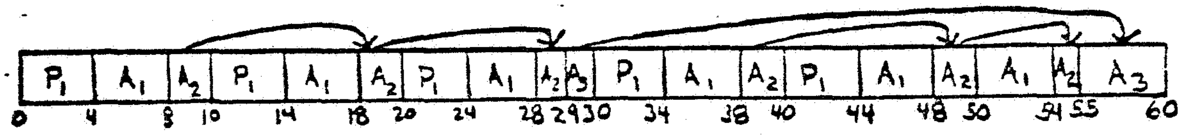
As before, the schedule is created iteratively beginning at the lowest level. With the exception of level r two schedules are created for each level i. The first schedule is for the interval t_s to D_i and is built upon the schedule for the interval t_s to D_{i-1} from the previous iteration concatenated with $(D_i-D_{i-1})/T_{i-1}$ copies of the second schedule at level i-1. This schedule is called $SHORT_i$ and SNP(j) denotes the number of j level primaries in the SHORT solution. The second schedule at level i is built on m_{i-1} copies of the second solution at level i-1 as in the previous algorithm. This schedule is called $FULL_{i-1}$ and FNP(j) denotes the number of j level primaries in the FULL solution. SMODIFY is a copy of MODIFY which creates SHORT solutions using the SNP(i) values. FMODIFY creates FULL solutions using FNP(i).

The following algorithm is executed whenever P_s succeeds:

<u>Algorithm 3.2</u>

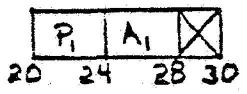
```
EXA_s := A_s

EXP_s := P_s

full idle time := T_1

for i := 1 to r do
   FNP(i) := 0
   fins := (D_i - D_{i-1})/T_i

   for j := 1 to r-1 do
      SNP(j) := SNP(j) + fins * FNP(j)
      end
   short idle time := short idle time + fins * full idle time

   create SHORT_i by SMODIFY(A_i-EXA_i, A_i+P_i-EXA_i-EXP_i, i)

   if D_i < D_r

   then begin
      for j := 1 to i-1 do
         FNP(j) := FNP(j) * m_{i-1}
         end
      full idle time := full idle time * m_{i-1}

      create FULL_i by FMODIFY(A_i, P_i, i)
      end
   end
```

Consider the use of this algorithm in the previous example. Recall that A_1=4, P_1=4, T_1=10, A_2=5, P_2=7, T_2=30, A_3=6, P_3=8 and T_3=60. The following schedule was produced for these jobs by Algorithm 3.1:

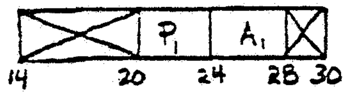


At time 14, P_1 succeeded so s=1, EXA_2=2 and EXP_1=4. All other EXA and EXP values are 0. D_1=20, D_2=30, D_3=60, R_1=6, R_2=16 and R_3=46.

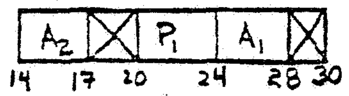
For i = s = 1, Algorithm 3.2 produces 6 units of IDLE for the SHORT schedule and the following FULL schedule:



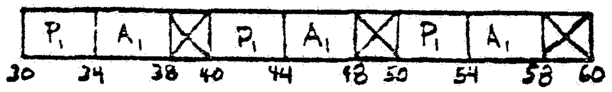
For i = 2, the schedule sent to SMODIFY is:



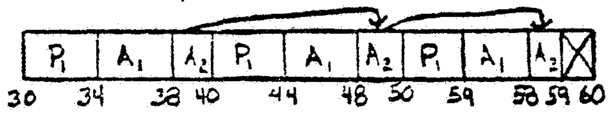
The $SHORT_2$ schedule is:



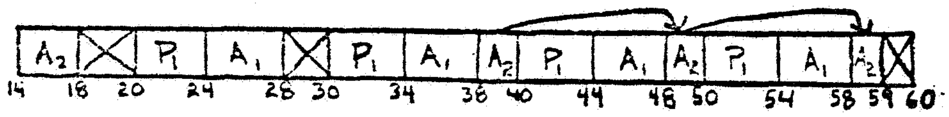
The schedule sent to FMODIFY is:



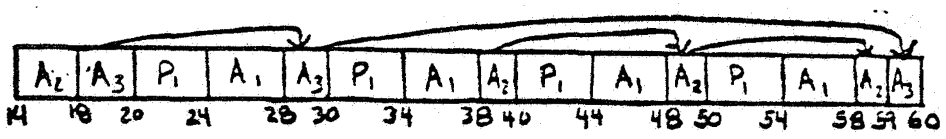
The $FULL_2$ schedule is:



For i = 3, the schedule sent to SMODIFY is:



The SHORT$_3$ schedule is:



The net effect of the new algorithm on this example is to add the execution of a P_1 in the interval 50-60.

Theorem 3.3: The schedule produced by Algorithm 3.2 is f-t optimal and has as much idle time as any f-t optimal schedule.

Proof: follows from Theorem 2.1.

Theorem 3.4: Algorithm 3.2 creates a f-t schedule in $O(M_{r-1})$ time.

Proof: follows from Theorem 2.2.

The reschedule algorithm produces a schedule for the period t_s to D_r which has at least as many primaries scheduled in the period as any other schedule which guarantees fault-tolerance given that the events occurring between t_0 and t_s have already occurred.

In the course of executing the schedule some idle time may be encountered in the schedule. Consider the following courses of action:

1. swap the idle time with some portion of a higher level task which is already scheduled.
2. execute part of an unscheduled primary.

Clearly either of these techniques may result in a larger number of primaries being executed than would result by leaving the time idle. Whether either method does increase the number of primaries depends on the run-time behavior (i.e. primary successes or failures) of the system.

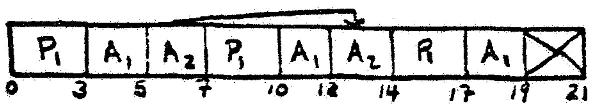
Consider a situation where idle time is to be filled by either of the above methods. With method 1 some heuristic must be used to decide which tasks to swap with the idle time. The heuristic may use probabilities of primary success (if they are known), potential saved alternate time, or other measures to make the decision, but it cannot predict which swapping will result in the best improvement. With the second method a heuristic is needed to decide among several possible partial executions. It may be useful to use these techniques but one can not predict which of the methods will yield the best result.

Given a set of jobs J an initial fault-tolerant schedule can be created by Algorithm 3.1. The jobs can then be executed as scheduled. When a primary algorithm succeeds a new schedule can be created which may allow more primaries to be executed. In all cases, the schedule produced includes as many primaries as any other schedule which guarantees that the deadlines will be met.

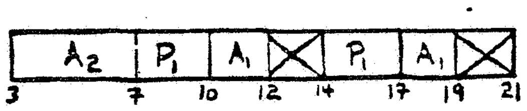
4 A Real-Time Fault-tolerant Schedule.

Algorithms 3.1 and 3.2 produce desirable schedules, however the execution time of the dynamic algorithm would be prohibitive in a real-time system. A precomputed schedule tree can be used as a real-time fault-tolerant schedule. The tasks executed in this schedule tree are exactly those executed by the above algorithms.

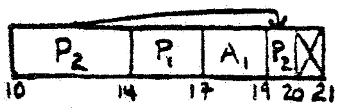
Consider the following example: Let J_1 and J_2 denote jobs such that $A_1=2$, $P_1=3$, $T_1=7$ and $A_2=4$, $P_2=5$, $T_2=21$. The schedule produced by Algorithm 3.1 is:



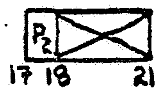
The execution of this schedule is simulated with the assumption that the scheduled primaries always succeed. P_1 succeeds at t=3 and Algorithm 3.2 (the reschedule algorithm) produces:



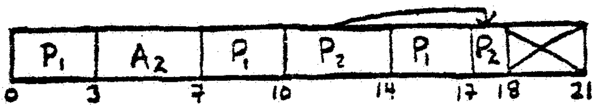
P_1 succeeds at t=10 and the reschedule algorithm produces:



P_1 succeeds at t=17 and the reschedule algorithm produces:



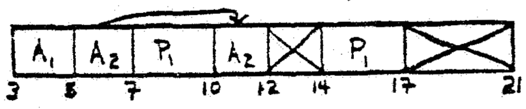
P_2 succeeds at t=18. The actually executed schedule is:



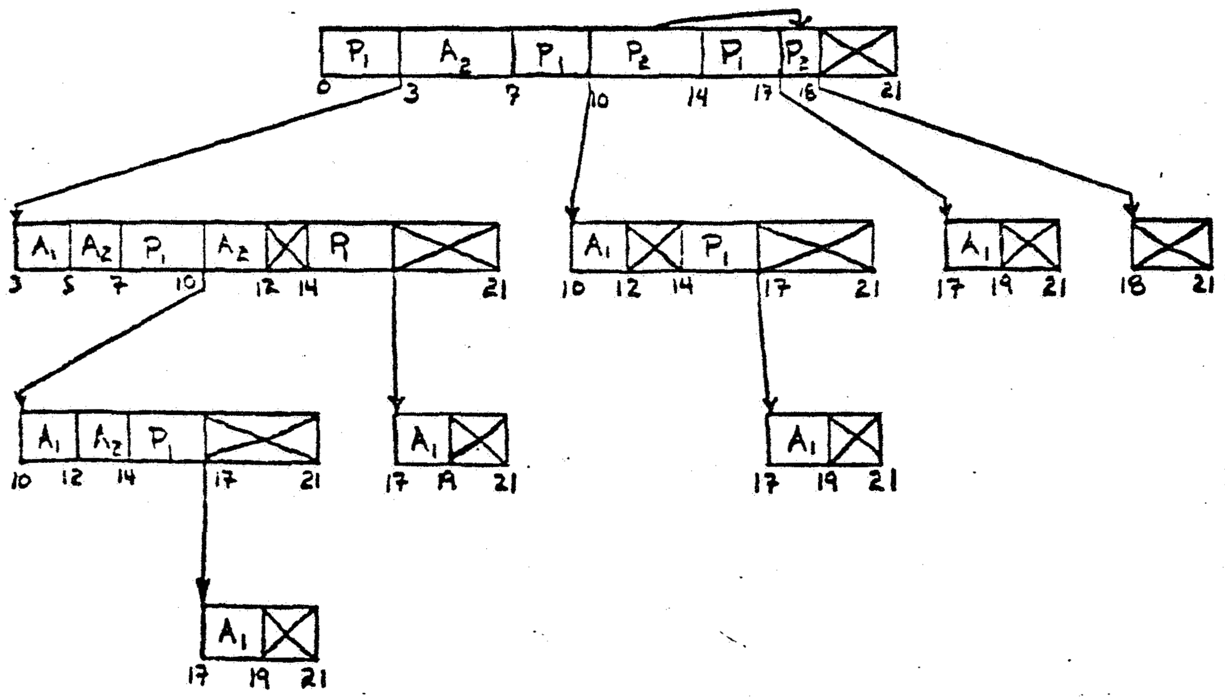
Use aes$<s_1s_2...s_k>$ to denote the actually executed schedule when the i^{th} primary succeeds if $s_i=1$ and fails if $s_i=0$. Thus the above schedule is aes<1111>. Denote the schedule produced by the reschedule algorithm after a series $<s_1s_2...s_k>$ of successes and failures with the notation res$<s_1s_2...s_k>$. Note that s_k will always be 1 when such a schedule exists. Let res<> denote the initial static schedule.

In the event that the primary failure rate is very low, it is advantageous to use the aes<111...1> schedule as "the schedule" for the system. This saves the overhead of rescheduling when a primary succeeds. "Backup" schedules are necessary to insure fault-tolerance in the unlikely event that a primary should fail.

Consider the specific case that P_1 fails at time t=3 in the above schedule. Using the remainder of the static schedule (res<>) would guarantee that all deadlines would be met including that for J_1 at t=7. On the other hand, the "actually executed" schedule aes<0111> from t=3 to t=21 could be used. This would guarantee that all deadlines would be met as long as no other primary should fail. The aes<0111> schedule is:

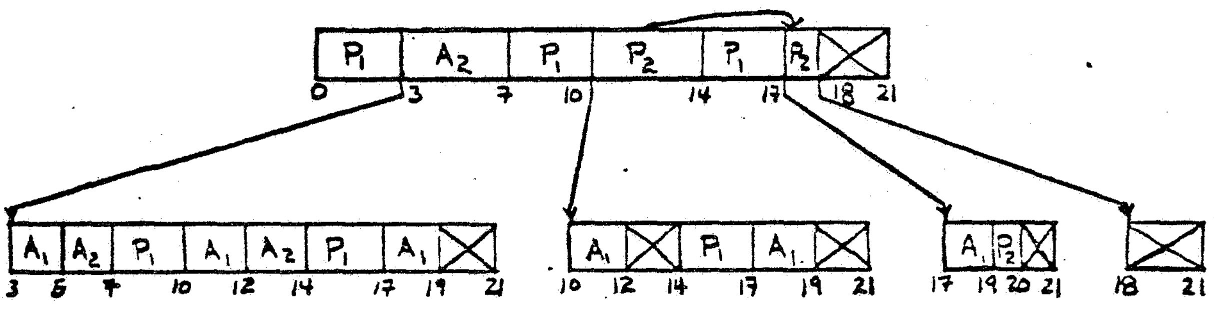


Note that the last 1 in aes<0111> is superfluous since only 3 primaries are executed. The following schedule is proposed for the above example:



The schedule at the root of the tree is executed until a primary fails. When a primary fails the corresponding schedule in the next level down in the tree is executed. Using this mechanism, the scheduling can be done in real time if this tree is constructed ahead of time.

If the tree is too large, it is pruned in the following manner: Consider a node in the tree which is the schedule to be executed if a failure occurs at time t_s in aes<$s_1s_2...s_k$>. If s_p is the rightmost 0 in $s_1s_2...s_k$ and s_j is the rightmost 1 in $s_1s_2...s_p$ replace the node with that portion of res<$s_1s_2...s_j$> from t_s to D_r. If no such p or j exists replace the node with the portion of res<> from t_s to D_r. The above example could be shortened to:



The sons of aes<1111> have all been replaced. The schedules aes<011>, aes<101>, aes<110>, and aes<1110> have been replaced by parts of res<>, res<1>, res<11>, and res<111>, respectively. Execution of this schedule proceeds the same way as before except that when executing a res schedule the schedule is executed to the end regardless of primary failures. The presence of alternates in the schedule will maintain fault-tolerance although some time may be wasted executing unnecessary alternates.

Using the mechanism presented in this chapter we can gain the benefits of the dynamic algorithm in a real-time system. The schedule tree mechanism schedules exactly the same tasks as scheduled by the Chapter 3 algorithms and if it is too large to store it can be pruned to an appropriate size with some degradation of performance.

5 Summary.

A scheduling algorithm was presented to maximize the number of primaries scheduled for a set of jobs with simply periodic requests. A modification of the algorithm was given which produces a static fault-tolerant optimal schedule for the jobs. Another modification of the algorithm was given to reschedule the remaining time when a primary success creates new idle time. Finally a schedule tree mechanism was described to gain the benefits of these scheduling algorithms in a real-time system.

6 Acknowledgements.

We would like to thank David A. Plaisted and Robert B. Kolstad for useful suggestions.

7 References.

[Campbell, et al., 79] Campbell, R. H., K. H. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism", Proceedings of the 1979 International Symposium on Fault-Tolerant Computing, June, 1979.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1010	2.	3. Recipient's Accession No.
4. Title and Subtitle A FAULT-TOLERANT SCHEDULING PROBLEM			5. Report Date February 1980
			6.
7. Author(s) A. L. Liestman and R. H. Campbell			8. Performing Organization Rept. No. R-80-1010
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NSG 1471
12. Sponsoring Organization Name and Address National Aeronautical and Space Administration Hampton, Virginia			13. Type of Report & Period Covered technical
			14.

15. Supplementary Notes

16. Abstracts

A real-time system provides a service which meets a set of specifications including real-time constraints. It is desirable to guarantee reliability in a real-time system. Reliability is a measure of how well the system conforms to its specifications. One technique used to improve reliability is fault-tolerance which incorporates redundancy into the system design. This redundancy is combined with error detection and error confinement techniques to prevent isolated failures from causing system failure. A deadline mechanism [Campbell, et al., 79] has been proposed to provide fault-tolerance in real-time systems. In this mechanism two independent algorithms are provided for each service subject to a deadline. An algorithm produces a fault-tolerant schedule for such a real-time system.

17. Key Words and Document Analysis. 17a. Descriptors

scheduling
fault-tolerance
reliability
deadline mechanism

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 26
	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71